Lower Bound Analysis

$$\log n! \leq \log n^n = n \log n.$$

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} \geq \frac{1}{2}(n \log n - n).$$

- So, $\log n! = \Theta(n \log n)$.
- Using the decision tree model, what is the average depth of a node?
- This is also $\Theta(\log n!)$.

$\log n-$ (1 or 2).

A Search Model (1)

Problem:
Given:

- A list L, of n elements
- A search key X

Solve: Identify one element in L which has key value X, if any exist.

Model:

- The key values for elements in L are unique.
- One comparison determines $<, =, >$.
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

What if the key values are not unique? Probably the cost goes down, not up. This is an assumption for *analysis*, not for implementation.

We would have a slightly different model (though no asymptotic change in cost) if our only comparison test was $<$. We would have a very different model if our only comparison was $= / \neq$.

A comparison-based model.

String data might require comparisons with very different costs.

A Search Model (2)

Goal: Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in L costs the same (array).

Is this a reasonable model and goal?

- We are assuming that the # of comparisons is proportional to runtime.
- Might not always share an array (assumption that all accesses are equal). For example, linked lists.
- We assume there is no relationship between value X and its position.

Linear Search

General algorithm strategy: Reduce the problem.

- Compare X to the first element.
- If not done, then solve the problem for $n-1$ elements.

```
Position linear_search(L, lower, upper, X) {
  if L[lower] = X then
    return lower;
  else if lower = upper then
    return -1;
  else
    return linear_search(L, lower+1, upper, X);
}
```

What equation represents the worst case cost?

$$f(n) = \begin{cases} 1 & n = 1 \\ f(n-1) + 1 & n > 1 \end{cases}$$

Lower Bound on Problem

Theorem: Lower bound (in the worst case) for the problem is n comparisons.

Proof: By contradiction.

- Assume an algorithm A exists that requires only $n-1$ (or less) comparisons of X with elements of L.
- Since there are n elements of L, A must have avoided comparing X with $L[i]$ for some value i.
- We can feed the algorithm an input with X in position i.
- Such an input is legal in our model, so the algorithm is incorrect.

Is this proof correct?

Be careful about assumptions on how an algorithm might (must) behave.
After all, where do new, clever algorithms come from? From different behavior than was previously assumed!

Fixing the Proof (1)

Error #1: An algorithm need not consistently skip position i.
Fix:

- On any given run of the algorithm, *some* element i gets skipped.
- It is possible that X is in position i at that time.

no notes

Fixing the Proof (2)

Error #2: Must allow comparisons between elements of L.
Fix:

- Include the ability to "preprocess" L.
- View L as initially consisting of n "pieces."
- A comparison can join two pieces (without involving X).
- The total of these comparisons is k.
- We must have at least $n-k$ pieces.
- A comparison of X against a piece can reject the whole piece.
- This requires $n-k$ comparisons.
- The total is still at least n comparisons.

no notes

Average Cost

How many comparisons does linear search do on average?

We must know the probability of occurrence for each possible input.

(Must X be in L?)

Ignore everything except the position of X in L. Why?

What are the $n+1$ events?

$$\mathbf{P}(X \notin L) = 1 - \sum_{i=1}^{n} \mathbf{P}(X = L[i]).$$

No, X might not be in L! What is this probability?

The actual values of other elements is irrelevent to the search routine.

$L[1], L[2], ..., L[n]$ and *not found*.

Assume that array bounds are $1..n$.

Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$.
Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let p_i be the probability that $X = L[i]$.
Let p_0 be the probability that $X \notin L[i]$ for any i.

$$
\begin{aligned}
f(n) &= k_0 p_0 + \sum_{i=1}^{n} k_i p_i \\
&= n p_0 + \sum_{i=1}^{n} i p_i
\end{aligned}
$$

What happens to the equation if we assume all p_i's are equal (except p_0)?

no notes

Computation

$$
\begin{aligned}
f(n) &= p_0 n + \sum_{i=1}^{n} ip \\
&= p_0 n + p \sum_{i=1}^{n} i \\
&= p_0 n + p \frac{n(n+1)}{2} \\
&= p_0 n + \frac{1-p_0}{n} \frac{n(n+1)}{2} \\
&= \frac{n+1+p_0(n-1)}{2}
\end{aligned}
$$

Depending on the value of p_0, $\frac{n+1}{2} \leq f(n) \leq n$.

$$p = \frac{1-p_0}{n}$$

.
Show a graph of p_0 vs. cost for $0 \leq p_0 \leq 1$, with y axis going from 0 to n.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Example: Quicksort variance is rather low. For this linear search, the variances is higher (normal curve).

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than X. Why?

Observation: If we look at $L[5]$ and find that X is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that X is bigger, then we know in one test that X is not in L. Great!

- What is wrong here?

We have more information a priori.

Can quit early.
What is best, worst, average cost? 1, n, $n/2$, respectively.
Effectively eliminates case of x not on list.

If we find that x is smaller, we only rule out one element.
Cost is 1 either way, but we don't get much information in worst case.
Small probability for big information, but big probability for small information.

Jump Search

Algorithm:

- From the beginning of the array, start making jumps of size k, checking $L[k]$ then $L[2k]$, and so on.
- So long as X is greater, keep jumping by k.
- If X is less, then use linear search on the last sublist of k elements.

This is called Jump Search.

What is the right amount to jump?

no notes

Analysis of Jump Search

- If $mk \leq n < (m+1)k$, then the total cost is at most $m + k - 1$ 3-way comparisons.

$$f(n,k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.$$

- What should k be?

$$\min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}$$

- Take the derivative and solve for $f'(x) = 0$ to find the minimum.
- This is a minimum when $k = \sqrt{n}$.
- What is the worst case cost?
  - Roughly $2\sqrt{n}$.

m is number of big steps, k is size of big step.

Lessons

We want to balance the work done while selecting a sublist with the work done while searching a sublist.

In general, make subproblems of equal effort.

This is an example of **divide and conquer**

What if we extend this to three levels?

- We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search
- While it might make sense to do a two-level algorithm (like jump search), it almost never makes sense to do a three-level algorithm
- Instead, we resort to recursion

This could lead us to binary search. It could also lead us to interpolation search.

Binary Search

```
int binary(int K, int* array, int left, int right) {
  // Return position of element (if any) with value K
  int l = left-1;
  int r = right+1;    // l and r beyond array bounds
  while (l+1 != r) {  // Stop when l and r meet
    int i = (l+r)/2;  // Middle of remaining subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```


$$f(n) = \begin{cases} 1 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Lower Bound (for Problem Worst Case)

How does n compare to $\sqrt{n}$ compare to $\log n$?

Can we do better?

Model an algorithm for the problem using a decision tree.

- Consider only comparisons with X.
- Branch depending on the result of comparing X with $L[i]$.
- There must be at least n leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Assumption: A deterministic algorithm: For a given input, the algorithm always does the same comparisons.

Since L is sorted, we already know the outcome of any comparisons between elements in L, so such comparisons are useless.

There must be some point in the algorithm, for each position in the array, where only that position remains as the possible outcome. Each such place corresponds to a (leaf) node.

Because a tree of n nodes requires at least this depth.

Average Cost of Binary Search (1)

An estimate given these assumptions:

- X is in L.
- X is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer k.

Cost?

- One chance to hit in one probe.
- Two chances to hit in two probes.
- 2^{i-1} to hit in i probes.
- $i \leq k$.

Average cost is $\log n - 1$.

no notes

Average Cost Lower Bound

- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length/n).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

(In worst case.)

Fill in tree row by row, left to right. So node i is at depth $\lfloor \log i \rfloor$.

Interpolation Search

(Also known as Dictionary Search) Search L at a position

that is appropriate to the value of X.

$$p = \frac{X - L[1]}{L[n] - L[1]}$$

Repeat as necessary to recalculate p for future searches.

That is, readjust for new array bounds.

Note that p is a fraction, so $\lfloor pn \rfloor$ is an index position between 0 and $n - 1$.

Quadratic Binary Search

This is easier to analyze:

- Compute p and examine $L[\lceil pn \rceil]$.
- If $X < L[\lceil pn \rceil]$ then sequentially probe
$$L[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, ...$$
until we reach a value less than or equal to X.
- Similar for $X > L[\lceil pn \rceil]$.
- We are now within $\sqrt{n}$ positions of X.
- ASSUME (for now) that this takes a constant number of comparisons.
- Now we have a sublist of size $\sqrt{n}$.
- Repeat the process recursively.
- What is the cost?

└Quadratic Binary Search

This is following the induction in a different way than Binary Search. Binary Search says break down list by (repeatedly) splitting in half. Interpolation search says break down list by (repeatedly) finding a square root-sized sublist.

We will come back and examine this assumption.

How many times can we take the square root of n?
Keep dividing the exponent by 2 until we reach 1 – that is, take the log of the *exponent*.
What is the exponent? It is $\log n$.
$\log \log n$ is the number of times that we can take the square root.

QBS Probe Count (1)

Cost is $\Theta(\log \log n)$ IF the number of probes on jump search is constant.

Number of comparisons needed is:

$$\sum_{i=1}^{\sqrt{n}} i\mathbf{P}(\text{need exactly } i \text{ probes})$$
$$= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}$$

This is equal to:

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

└QBS Probe Count (1)

no notes

QBS Probe Count (2)

$$\sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

$$\begin{aligned}
&= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\
&= (\mathbf{P}_1 + ... + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + ... + \mathbf{P}_{\sqrt{n}}) + \\
&\qquad (\mathbf{P}_3 + ... + \mathbf{P}_{\sqrt{n}}) + \cdots \\
&= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}
\end{aligned}$$

└QBS Probe Count (2)

no notes

QBS Probe Count (3)

We require at least two probes to set the bounds, so cost is:

$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes})$$

Useful fact (Čebyšev's Inequality):
The probability that we need probe i times ($\mathbf{P}_i$) is:

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2 n} \leq \frac{1}{4(i-2)^2}$$

since $p(1-p) \leq 1/4$.

This assumes uniformly distributed data.

└QBS Probe Count (3)

Original C's Inequality $\leq$ the result of recognizing that $p(1-p) \leq 1/4$.

Important assumption!

QBS Probe Count (4)

Final result:

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112$$

Is this better than binary search?

What happened to our proof that binary search is optimal?

The assumption of uniform distribution (resulting in constant number of probes on average) is much stronger than the assumptions used by the lower bounds proof.

Comparison (1)

Let's compare $\log \log n$ to $\log n$.

n	$\log n$	$\log \log n$	Diff
16	4	2	2
256	8	3	2.7
$64K$	16	4	4
2^{32}	32	5	6.4

Now look at the actual comparisons used.

- Binary search $\approx \log n - 1$
- Interpolation search $\approx 2.4 \log \log n$

n	$\log n - 1$	$2.4 \log \log n$	Diff
16	3	4.8	worse
256	7	7.2	$\approx$ same
$64K$	15	9.6	1.6
2^{32}	31	12	2.6

no notes

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?

Taking an interpolation point.

QBS